18｜控制结构：if的“快乐路径”原则

2021-11-22 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

讲述：Tony Bai
时长 13:34 大小 12.44M

你好，我是 Tony Bai。

1984 年图灵奖获得者、著名计算机科学家尼古拉斯·沃斯（Niklaus Wirth）提出过著名的“程序 = 数据结构 + 算法”的公式。在前面的课程中，我们花了很多时间讲解了 Go 语言的基本数据类型和复合数据类型，这些对应的就是公式中数据结构，通过这些数据类型我们可以建立起复杂的数据结构。

那么公式中的算法呢？算法是对真实世界运作规律的抽象，是解决真实世界中问题的步骤。在计算机世界中，再复杂的算法都可以通过顺序、分支和循环这三种基本的控制结构造出来。

顺序结构自然不用说了，我们要关注的主要是后面两个。所以，这一节课开始的连续三节课，我们都会聚焦于 Go 语言中的分支和循环这两种控制结构。

那么 Go 语言对分支与循环两种控制结构的支持是怎么样的呢？针对程序的分支结构，Go 提供了 if 和 switch-case 两种语句形式；而针对循环结构，Go 只保留了 for 这一种循环语句形式。这节课我们就先从 Go 语言分支结构之一的 if 语句开始讲起。

Go 中的分支结构之认识 if 语句

在01 讲中我们提到过，Go 语言是站在 C 语言等的肩膀之上诞生与成长起来的。Go 语言继承了 C 语言的很多语法，这里就包括控制结构。但 Go 也不是全盘照搬，而是在继承的基础上又加上了自己的一些优化与改进，比如：

> Go 坚持“一件事情仅有一种做法的理念”，只保留了 for 这一种循环结构，去掉了 C 语言中的 while 和 do-while 循环结构；
>
> Go 填平了 C 语言中 switch 分支结构中每个 case 语句都要以 break 收尾的“坑”；
>
> Go 支持了 type switch 特性，让“类型”信息也可以作为分支选择的条件；
>
> Go 的 switch 控制结构的 case 语句还支持表达式列表，让相同处理逻辑的多个分支可以合并为一个分支，等等。

如果你这个时候还不是很懂我提到的这些改进点，没有关系，在后面的几节课中，我会为你详细讲解 Go 关于控制结构的各个优化和改进点。

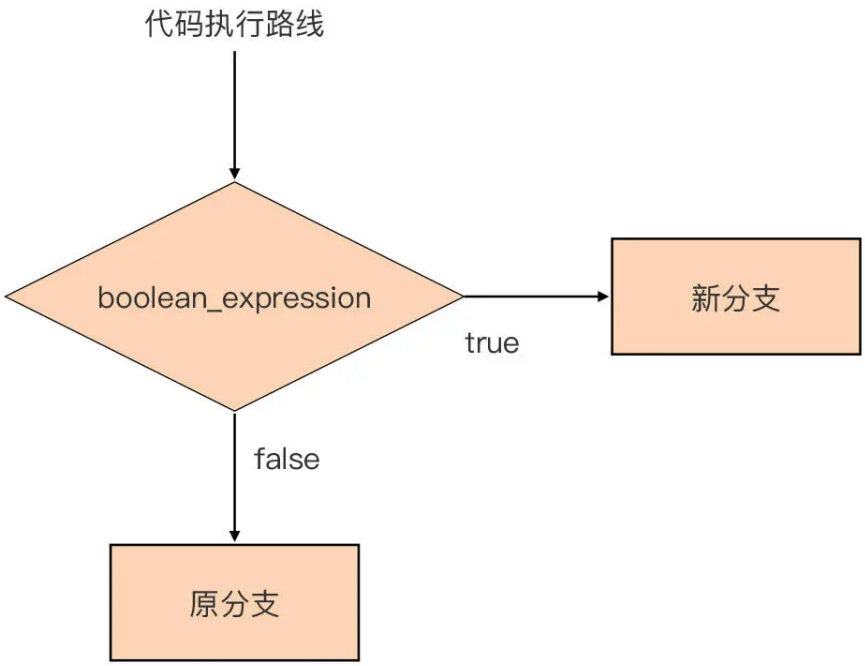
那么，Go 中的 if 语句又有什么创新点呢？我们先来认识一下 Go 中的 if 语句。

if 语句是 Go 语言中提供的一种分支控制结构，它也是 Go 中最常用、最简单的分支控制结构。它会根据**布尔表达式**的值，在两个分支中选择一个执行。我们先来看一个最简单的、单分支结构的 if 语句的形式：

复制代码

```
1 if boolean_expression {
2     // 新分支
3 }
4
5 // 原分支
```

分支结构是传统结构化程序设计中的基础构件，这个 if 语句中的代码执行流程就等价于下面这幅流程图：



从图中我们可以看到，代码执行遇到 if 分支结构后，首先会对其中的布尔表达式（boolean_expression）进行求值，如果求值结果为 true，那么程序将进入**新分支**执行，如果布尔表达式的求值结果为 false，代码就会继续沿着原分支的路线继续执行。

虽然各种编程语言几乎都原生支持了 if 语句，但 Go 的 if 语句依然有着自己的特点：

第一，和 Go 函数一样，if 语句的分支代码块的左大括号与 if 关键字在同一行上，这也是 Go 代码风格的统一要求，gofmt 工具会帮助我们实现这一点；

第二，if 语句的布尔表达式整体不需要用括号包裹，一定程度上减少了开发人员敲击键盘的次数。而且，if 关键字后面的条件判断表达式的求值结果必须是布尔类型，即要么是 true，要么是 false：

复制代码

```
1 if runtime.GOOS == "linux" {
2     println("we are on linux os")
```

```
3 }
```

如果判断的条件比较多，我们可以用多个逻辑操作符连接起多个条件判断表达式，比如这段代码就是用了多个逻辑操作符 && 来连接多个布尔表达式：

复制代码

```
1 if (runtime.GOOS == "linux") && (runtime.GOARCH == "amd64") &&
2     (runtime.Compiler != "gccgo") {
3     println("we are using standard go compiler on linux os for amd64")
4 }
```

除了逻辑操作符 && 之外，Go 还提供了另外两个逻辑操作符，我总结到了这张表里。

逻辑操作符	含义	表达式求值举例
&&	逻辑与	a && b： 当a和b都为true时，该表达式的求值结果为true
‖	逻辑或	a ‖ b：当a和b至少一个为true时，该表达式的求值结果为true
!	逻辑非	!a ： 若a为true，表达式求值结果为false；反之若a为false，表达式求值结果为true

极客时间

你可能也注意到了，上面示例代码中的每个布尔表达式都被小括号括上了，这又是什么原因呢？这是为了降低你在阅读和理解这段代码时，面对操作符优先级的心智负担，这也是我个人的编码习惯。

Go 语言的操作符是有优先级的。这里你要记住，一元操作符，比如上面的逻辑非操作符，具有最高优先级，其他操作符的优先级如下：

优先级（从高到底）	操作符列表
5	*、 / 、%、 << 、>> 、&、 &^
4	+、 −
3	!= 、==、 < 、<= 、> 、 >=
2	&&
1	‖

极客时间

操作符优先级决定了操作数优先参与哪个操作符的求值运算，我们以下面代码中 if 语句的布尔表达式为例：

复制代码

```
func main() {
    a, b := false,true
    if a && b != true {
        println("(a && b) != true")
        return
    }
    println("a && (b != true) == false")
}
```

执行这段代码会输出什么呢？你第一次读这段代码的时候，可能会认为输出 (a && b) != true，但实际上我们得到的却是 a && (b != true) == false。这是为什么呢？

这段代码的关键就在于，if 后面的布尔表达式中的操作数 b 是先参与 && 的求值运算，还是先参与!= 的求值运算。根据前面的操作符优先级表，我们知道，!= 的优先级要高于 &&，因此操作数 b 先参与的是!= 的求值运算，这样 if 后的布尔表达式就等价于 a && (b != true) ，而不是我们最初认为的 (a && b) != true。

如果你有时候也会记不住操作符优先级，不用紧张。从学习和使用 C 语言开始，我自己就记不住这么多操作符的优先级，况且不同编程语言的操作符优先级还可能有所不同，所以我个人倾向在 if 布尔表达式中，使用**带有小括号的子布尔表达式**来清晰地表达判断条件。

这样做不仅可以消除了自己记住操作符优先级的学习负担，同时就像前面说的，当其他人阅读你的代码时，也可以很清晰地看出布尔表达式要表达的逻辑关系，这能让我们代码的可读性更好，更易于理解，不会因记错操作符优先级顺序而产生错误的理解。

除了上面的最简形式，Go 语言的 if 语句还有其他多种形式，比如二分支结构和多（N）分支结构。

二分支控制结构比较好理解。比如下面这个例子，当 boolean_expression 求值为 true 时，执行分支 1，否则，执行分支 2：

复制代码

```
if boolean_expression {
  // 分支1
} else {
  // 分支2
}
```

多分支结构由于引入了 else if，理解起来稍难一点点，它的标准形式是这样的：

复制代码

```
if boolean_expression1 {
  // 分支1
} else if boolean_expression2 {
  // 分支2

... ...

} else if boolean_expressionN {
  // 分支N
} else {
  // 分支N+1
}
```

我们以下面这个四分支的代码为例，看看怎么拆解这个多分支结构：

复制代码

```
if boolean_expression1 {
    // 分支1
} else if boolean_expression2 {
    // 分支2
```

```
} else if boolean_expression3 {
    // 分支3
} else {
    // 分支4
}
```

要理解这个略复杂一些的分支结构，其实很简单。我们只需要把它做一下等价变换，变换为我们熟悉的二分支结构就好了，变换后的代码如下：

复制代码

```
if boolean_expression1 {
    // 分支1
} else {
    if boolean_expression2 {
        // 分支2
    } else {
        if boolean_expression3 {
            // 分支3
        } else {
            // 分支4
        }
    }
}
```

这样等价转换后，我们得到一个层层缩进的二分支结构，通过上面我们对二分支的分析，再来理解这个结构就十分容易了。

支持声明 if 语句的自用变量

无论是单分支、二分支还是多分支结构，我们都可以在 if 后的布尔表达式前，进行一些变量的声明，在 if 布尔表达式前声明的变量，我叫它 **if 语句的自用变量**。顾名思义，这些变量只可以在 if 语句的代码块范围内使用，比如下面代码中的变量 a、b 和 c：

复制代码

```
func main() {
    if a, c := f(), h(); a > 0 {
        println(a)
    } else if b := f(); b > 0 {
        println(a, b)
    } else {
        println(a, b, c)
    }
```

```
9 }
```

我们可以看到自用变量声明的位置是在每个 if 语句的后面，布尔表达式的前面，而且，由于声明本身是一个语句，所以我们需要把它和后面的布尔表达式通过分号分隔开。

这里又涉及到了代码块与作用域的概念，这是我们在第 11 讲中学习到的内容。如果你觉得概念有些模糊了，可以回过头去复习一下。根据第 11 讲中的讲解，我们知道，上面代码中声明的变量 a、b、c 都位于各级 if 的隐式代码块中，它们的作用域起始于它声明所在的代码块，并一直可扩展至嵌入到这个代码块的所有内层代码块中。

在 if 语句中声明自用变量是 Go 语言的一个惯用法，这种使用方式直观上可以让开发者有一种代码行数减少的感觉，提高可读性。同时，由于这些变量是 if 语句自用变量，它的作用域仅限于 if 语句的各层隐式代码块中，if 语句外部无法访问和更改这些变量，这就让这些变量具有一定隔离性，这样你在阅读和理解 if 语句的代码时也可以更聚焦。

不过前面我们第 11 讲也重点提到过，Go 控制结构与短变量声明的结合是“变量遮蔽”问题出没的重灾区，你在这点上一定要注意。

到这里，我们已经学过了 if 分支控制结构的所有形式，也了解了 if 语句通过短变量声明形式声明自用变量的优点与不足。那么在日常开发中，这些 if 分支控制结构形式是随意使用的吗？有什么优化方案吗？

if 语句的“快乐路径”原则

我们已经学了 if 分支控制结构的三种形式了，从可读性上来看，单分支结构要优于二分支结构，二分支结构又优于多分支结构。那么显然，我们**在日常编码中要减少多分支结构，甚至是二分支结构的使用，这会有助于我们编写出优雅、简洁、易读易维护且不易错的代码。**

我们用一个具体的例子直观地体会一下我的这个观点，下面是两段逻辑相同但形式不同的伪代码段：

复制代码

```
1 //伪代码段1：
2
```

```
func doSomething() error {
  if errorCondition1 {
    // some error logic
    ... ...
    return err1
  }

  // some success logic
  ... ...

  if errorCondition2 {
    // some error logic
    ... ...
    return err2
  }

  // some success logic
  ... ...
  return nil
}

// 伪代码段2：

func doSomething() error {
  if successCondition1 {
    // some success logic
    ... ...

    if successCondition2 {
      // some success logic
      ... ...

      return nil
    } else {
      // some error logic
      ... ...
      return err2
    }
  } else {
    // some error logic
    ... ...
    return err1
  }
}
```

即便你是刚入门的 Go 新手，你大概也能看出上面代码的优劣。

我们看看只使用了单分支控制结构的伪代码段 1，我们看到代码段 1 有这几个特点：

没有使用 else 分支，失败就立即返回；

“成功”逻辑始终“居左”并延续到函数结尾，没有被嵌入到 if 的布尔表达式为 true 的代码分支中；

整个代码段布局扁平，没有深度的缩进；

而另外一个实现了同样逻辑的伪代码段 2，就使用了带有嵌套的二分支结构，它的特点如下：

整个代码段呈现为“锯齿状”，有深度缩进；

“成功”逻辑被嵌入到 if 的布尔表达式为 true 的代码分支中；

很明显，伪代码段 1 的逻辑更容易理解，也更简洁。Go 社区把这种 if 语句的使用方式称为 if 语句的“**快乐路径**（Happy Path）”原则，所谓“快乐路径”也就是成功逻辑的代码执行路径，它的特点是这样的：

仅使用单分支控制结构；

当布尔表达式求值为 false 时，也就是出现错误时，在单分支中快速返回；

正常逻辑在代码布局上始终“靠左”，这样读者可以从上到下一眼看到该函数正常逻辑的全貌；

函数执行到最后一行代表一种成功状态。

Go 社区推荐 Gopher 们在使用 if 语句时尽量符合这些原则，如果你的函数实现代码不符合“快乐路径”原则，你可以按下面步骤进行重构：

尝试将“正常逻辑”提取出来，放到“快乐路径”中；

如果无法做到上一点，很可能是函数内的逻辑过于复杂，可以将深度缩进到 else 分支中的代码析出到一个函数中，再对原函数实施“快乐路径”原则。

小结

好了，今天的课讲到这里就结束了，现在我们一起来回顾一下吧。

分支控制结构是构造现实中复杂算法的三大基础控制结构之一，Go 语言通过 if 与 switch 语句对分支控制结构提供了支持。在这节课中，我们重点讲解了 if 语句，我建议你记住以下几点：

第一，if 语句是 Go 语言中最常用的分支控制语句，也是最简单的分支控制结构。if 语句通过对布尔表达式的求值决定了后续代码执行要进入的哪条分支。当需要复杂条件判断时，我们可以使用逻辑操作符连接多个布尔表达式，作为 if 语句的判断条件表达式。如果这么做了，我们还要注意各个操作符的优先级，我个人建议尽量用小括号对各个布尔表达式进行清晰地隔离，这样可以提升代码可读性。

第二，Go 的 if 语句提供了多种使用形式，包括单分支、双分支以及多分支。多分支理解起来略有难度，我们可以将它等价转换为双分支来理解。

第三，if 语句支持在布尔表达式前声明自用变量，这些变量作用域仅限于 if 语句的代码块内部。使用 if 自用变量可以一定程度简化代码，并增强与同函数内其他变量的隔离，但这也十分容易导致变量遮蔽问题，你使用时一定要注意。

最后一点，if 语句的三种使用形式的复杂度与可读性不一，我们建议在使用 if 语句时尽量符合“快乐路径”原则，这个原则通常只使用最容易理解的单分支结构，所有正常代码均“靠左”，这让函数内代码逻辑一目了然，提升了代码可读性与可维护性。

思考题

今天，我依然出了一个思考题：如果一个 if 语句使用了多分支结构，如下面代码这样，那么 if 语句中的几个布尔表达式如何排列能达到最好的效果呢？

提示一下，几个布尔表达式能够被命中的概率是不同的，你在答案中可以自行假设一下。期待在留言区看到你的分析。

复制代码

```
func foo() {
    if boolean_expression1 {

    } else if boolean_expression2 {

    } else if boolean_expression3 {

```

```
8       } else {
9
10      }
11  }
```

欢迎你把这节课分享给更多对 Go 语言中的 if 语句感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 4 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 17 | 复合数据类型：用结构体建立对真实世界的抽象

下一篇 19 | 控制结构：Go的for循环，仅此一种

更多学习推荐

精选留言 (11)

写留言

Darren 民

2021-11-22

如果加上“布尔表达式3在这段代码中实际被命中的机会更多，布尔表达式2次之，布尔表达式1最少”，这个条件，那么最优的性能最好的写法应该是最大概率的放到最前面，因此可以改成如下：

...

展开

作者回复: 厉害！👍

共 7 条评论 > 9

Darren 民

2021-11-22

可以改成这样子吧

```
func foo() {
    if boolean_expression1 {
...
```

展开

作者回复: 改成快乐路径方式是ok的。

可能思考题没说清楚。这里说的“效果最好”，指的是这段代码的执行性能最好。

提示一下：如果从统计概率而言，布尔表达式3在这段代码中实际被命中的机会更多，布尔表达式2次之，布尔表达式1最少，那么排列应该如何变
化才能让这段代码执行性能最好呢？

共 2 条评论 > 5

lesserror

2021-11-23

对于if的“快乐路径”原则深表认同，每次看见同事写的四五层以上的 if else 语句，有种“想死” 的冲动。

自以为逻辑能力很强，殊不知自己一时爽，后来者却无法维护这样的代码了。

展开

作者回复: 是的。尽量写简洁易读的代码才是王道。

3

用0和1改变自己

2021-11-23

把命中率高的依次放前面，性能会好些。需要注意的是，并不是所有if都遵从快乐原则，毕竟很多逻辑只是简单确定一个值，return的逻辑在下面。Happy Path还是更是适合错误判断，而不是单纯选择

共 1 条评论 >

1

aoe

2021-11-23

跟着老师的风格走，多个 if 条件组合的时候加上括号提高可读性。快乐路径很有意思，比 Fail-fast 更吸引眼球

展开

1

进化菌

2021-11-22

if快乐路径，其实就是减少不必要的嵌套，让代码结构简单明了。
思考题里，条件的优先级应该是命中率高的排前面，else 有时候似乎没太必要写。

1

qinsi

2021-11-22

还是之前的问题，happy path似乎让comma ok不再简洁

comma ok:
```go
m := map[string]int {...
```

展开

作者回复: 相对于第一个深层嵌套的“不易读”，下面虽然verbose一些，但代码一目了然啊。不知你是否有同样感觉。

共 4 条评论 > 1

Forest
2021-11-22

使用switch case ;
func foo() {
switch expression {
case condition1:
...
展开

作者回复: 可能思考题没说清楚。这里说的“效果最好”，指的是这段代码的执行性能最好。

提示一下：如果从统计概率而言，布尔表达式3在这段代码中实际被命中的机会更多，布尔表达式2次之，布尔表达式1最少，那么排列应该如何变化才能让这段代码执行性能最好呢？

btw，使用switch-case，说明你基础很好。

1

My.life
2021-11-22

func main() {
a, b := true, false
if a && b != true {
println("(a && b) != true")...
展开

作者回复: 笔误，后续编辑会改一下。不影响后续的理解。感谢指出。

1

罗杰
2021-11-23

我在觉得记运算符优先级没有必要，按照自己的需求去括号括起来，大家的理解都不会有误差

作者回复: 是这样的。

多选参数

2021-11-23

“如果从统计概率而言，布尔表达式3在这段代码中实际被命中的机会更多，布尔表达式2次之，布尔表达式1最少”这句话中从统计概率而言？应该说假如从实际执行情况中发现布尔表达式 3 被执行的更多？或者通过分析，发现条件 3 在更多的场景下容易满足？

展开

作者回复: 你理解的对。